Enterprise Architektur-Muster

JULIAN BRUDER*, ABDELLAH FILALI*, and LUCA FRANKE*, Hochschule für Technik, Wirtschaft und Kultur Leipzig (HTWK Leipzig), Deutschland

Blah abstrakt…

1 EINLEITUNG

Mit E-Commerce-Beispiel motivieren

2 GRUNDLAGEN VON ENTERPRISE-ARCHITEKTUREN

Verteilte Systeme …
Architekturen …
Komponenten …
…

3 KLASSISCHE ENTERPRISE-ARCHITEKTUREN

3.1 Service-oriented

Die Service-orientierte Achitektur (SOA) basiert auf die Verwendung von Diensten - sogenannten Services. Ein Dienst kapselt eine Funktionalität und stellst diese über eine fest definirte Schnittstelle bereit [Bianco et al. 2007][S. 3]. Dieses kann aus verschienen Diesten bestehen. Betrachten wir im Folgenden die Hauptbestandteile der SOA [Endrei et al. 2004]:

- Service Provider: Bietet einen spezifischen Dienst an
- Service Bus: Dienst, das die Orchestierung zwischen Service Requester und Service Provider gewährleistet.
- Service Consumer: Nutzt einen bereitgestelten Dienst, dieses kann einen End-User oder ein anderer Dienst sein.
- Service Registry: Dient als zentrales Verzeichnis, die Informationen über die verfügbare Services speichert.

Ein Service Provider registriert die Informationen seines Diensts in den Service Registry. Die speicherte Informationen sind der Interface-Vertrag und der Endpoint des Services. Der Endpoint stellt die Schnittstelle zum Dienst dar, währen der Interface-Vertrag des Service Provider festlegt, wie der Service Consumer den Dienst nutzen kann.

Der Service Consumer kann dann über den Broker den benötigten Service im Service Registry finden und den Service Provider gemäß dem klar definierten Vertrag nutzen.

Ein Service Broker ist für die Komunikation zwischen ein Service Consumer und ein Service Provider zuständig, während ein Service Bus, insbesondere ein Enterprise Service Bus, kann als ein erweiteres Service Broker angesehen werden. Dieses ist nicht nur dafür verantowortlich, für die Nachrichtvermitlung zwischen Diensten , sondern auch für die Steuerung und die Übersetzung aller Nachrichten, unabhängig davon welches Nachrichtenprotokoll verwendet wird [Endrei et al. 2004].

Die Abbildung 1 fasst diese Beziehung zusammen.

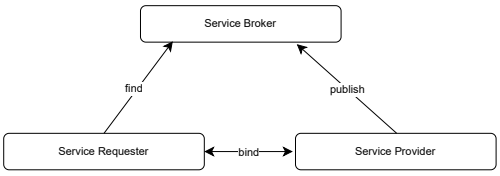

Abb. 1. Aufbau Service-oriented Architecture

Da jeder Service einen spezifischen Dienst anbieten und diese durch eine klar definirte Schnitstellen, können Dienste unabhängig von einandere weiterentwickelt werden, was zu einer höheren agilität führt. Außerdem neue Dienste erfüllen einen klar abgegrenzten und diese kann in rahmen in

Dienste können an anderen Stellen in der Software erneut eingesetzt oder mit anderen Diensten kombiniert werden, um die Bedürfnisse eines Unternehmens zu erfüllen. Dies führt zu weniger Duplikaten von Ressourcen, einer höheren Wiederverwendbarkeit von Code und reduzierten Kosten [Endrei et al. 2004].

Insgesamt weist die Service-oriented Architecture also eine sehr hohe Agilität auf.

Betrachten wir erneut das E-Commerce-Beispiel aus der Einleitung. Dafür definieren wir drei Arten von Services:

- `OrderService`: Ein Dienst, das den gesamenten Bestellvorgang initiiert
- `PaymentService`: Ein Dienst, das der Bezahlvorgang startet
- `ShipmentService`: Ein Dienst, das der Versandvorgang startet

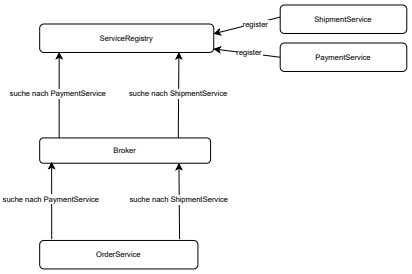

Abb. 2. E-Commerce-Beispiel mit Service-oriented Architecture

Wie Abbildung 2 zeigt, die Dienste`ShipmentService` und `PaymentService` werden als externe Dienste betrachtet, die ihre Informationen (Name, Bescheibung des Dienstes und Endpoint) bei dem Broker veröffentlichen. Der `OrderService` ist ein interner Dienst, der die beiden externen Dienste verwendet. Dies erfolgt durch Anfragen an den Broker, der die entsprechenden Endpoints der Dienste ermittelt, sodass der `OrderService` über HTTP-Requests auf die benötigten Dienste zugreifen kann.

*Alle Studierenden trugen zu gleichen Teilen zu dieser Arbeit bei.

Diese Arbeit wurde im Rahmen des Mastermoduls „Software Engineering“ (Dozent: Prof. Dr. Andreas Both) an der HTWK Leipzig im Wintersemester 2024/2025 erstellt.

Das Beispiel zeigt, wie die Service-oriented Architecture die Agilität fördern kann. Externe Dienste können ohne große Anpassungen integriert werden und in internen Dienste kombiniert und verwendet werden. Neue Entwickler können an einzelnen Diensten arbeiten, ohne das gesamte System verstehen zu müssen, da die Dienste über definierte Schnittstellen miteinander kommunizieren.

Die vollständige Implementierung des E-Commerce-Beispiels ist bei GitHub [1] zu finden.

4 MODERNE ENTERPRISE-ARCHITEKTUREN

4.1 Event-Driven Architecture

Die Event-Driven Architecture wählt als Basis einen anderen Ausgangspunkt als die bisherigen Architekturmuster. Während bei letzteren Komponenten Dienste bereitstellen, welche von anderen Komponenten explizit genutzt werden, verhalten sich Dienstbereitstellende Komponenten in der Event-Driven Architecture reaktiv, werden also implizit von Dienst-konsumierenden Komponenten genutzt [Garlan and Shaw 1994]. Ein System reagiert somit asynchron auf Zustandsänderungen, also Ereignisse in diesem System [Manchana 2021]. Die in dieser Architektur minimalen Einheiten, welche Informationen einer Zustandsänderung kapseln, werden *Events* genannt. Die Idee der impliziten Behandlung von Ereignissen ist nicht neu und taucht erstmals 1994 im von Garlan und Shaw publizierten Papier *„An introduction to Software Architecture“* auf.

Betrachten wir im Folgenden die Basis-Bestandteile der Event-Driven Architecture:

- Ereignis (englisch *Event*): Kapselt Information einer Zustandsänderung eines Systems
- Produzent (englisch *Producer*): Komponente, die Event erzeugt
- Herausgeber (englisch *Publisher*): Komponente, die, von Produzenten erzeugte, Events publiziert
- Konsument (englisch *Consumer*): Komponente, die auf publizierte Events reagiert
- Vermittler (englisch *Mediator*): Komponente zwischen Produzenten und Konsumenten - filtert Events und verteilt diese auf Konsumenten
- Event-Bus: Oft auch *Event-Broker* genannt - bietet die Infrastruktur für die Gesamtheit der Vermittler

Abstrakt kann ein Event als Vertrag zwischen Produzenten und Konsumenten am Event-Bus betrachtet werden. Der Konsument nutzt die Spezifikation des Events am Bus, der Produzent implementiert jene Spezifikation. Abbildung 3 stellt diesen Vertrag dar.

Durch den Vertrag weisen die Events am Event-Bus starke Kohäsion und somit lose Kopplung auf. Diese lose Kopplung minimiert nicht nur kaskadierende Fehler, sondern ermöglicht agilen Entwickler-Teams durch klar abgegrenzte Features einfach definierbare Iterationen - eine Menge von Events, deren Erzeugung und Konsumierung.

Weiter sind Events oft nah an dem, was Ereignisse in realen Prozessen sind, also domain-driven. Gebündelt ermöglichen obige Punkte die kontinuierliche Auslieferung von Software in kurzen Intervallen.

[1] https://github.com/Beleg-6-EAP/demo-soa-ecommerce

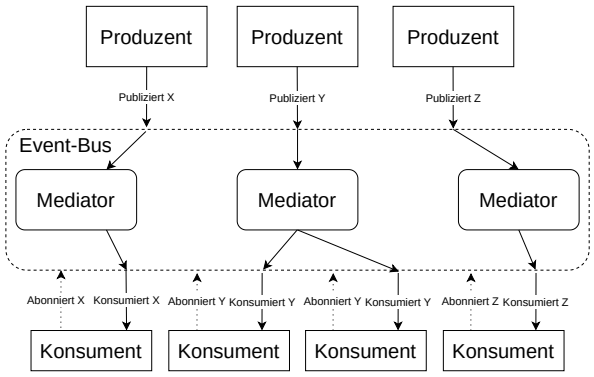

Abb. 3. Vertrag zwischen Produzenten und Konsumenten am Event-Bus

Außerdem garantiert die asynchrone Behandlung von Ereignissen zusammen mit der loosen Kopplung maximale Skalierung. Daher sind Event-Driven Architekturen besonders für datenintensive Echtzeit-Anwendungen wie IoT (Internet of Things) und Analytics geeignet [Siddiqui et al. 2023].

Betrachten wir erneut das E-Commerce-Beispiel aus der Einleitung. Dafür definieren wir drei Arten von Events:

- `OrderCreated`: Ein Event, das genau dann erzeugt wird, wenn eine neue Bestellung aufgegeben wird
- `PaymentProcessed`: Ein Event, das genau dann erzeugt wird, wenn der Bezahlvorgang abgeschlossen wird
- `ShipmentInitiated`: Ein Event, das genau dann erzeugt wird, wenn die Bestellung versandt wird

Weiter teilen wir die Funktionalität ähnlich wie bei der Microservice-Architektur in die drei verschiedenen Dienste `OrderService`, `PaymentService` und `ShipmentService` auf.

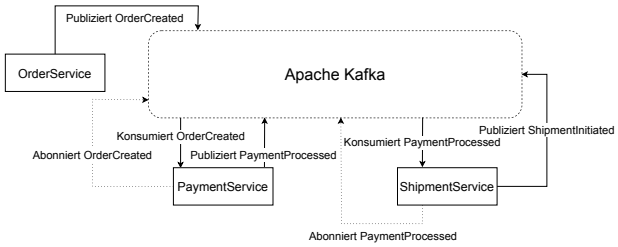

Abb. 4. E-Commerce-Beispiel mit Event-Driven Architecture

Wie Abbildung 4 zeigt, sind alle drei Dienste Produzenten und Publisher, erzeugen also Events und veröffentlichen diese. Die Dienste `PaymentService` und `ShipmentService` sind zudem Konsumenten, sodass ersterer auf Events des Typs `OrderCreated` und zweiterer auf Events des Typs `ShipmentInitiated` reagiert. Eine beispielhafte Implementierung des `PaymentService` mit Apache Kafka als Event-Broker ist im Anhang A.1 zu finden. Die vollständige Implementierung des E-Commerce-Beispiels ist bei GitHub [2] zu finden.

[2] https://github.com/Beleg-6-EAP/demo-eda-ecommerce

Das Beispiel zeigt, dass die Event-Driven Architektur mit weiteren agilen Strukturen wie Microservices kombiniert werden kann, was die Agilität der Architektur weiter erhöht. Die damit einhergehende Komplexität stellt teilweise hohe Anforderungen an die Entwickler. Aufgrund der Asynchronität der Behandlung von Ereignissen ist die Testung des Systems meist schwer und die Fehlerbehandlung essentiell. Mögliche Problemquellen schließen dabei unter anderem Event-Verlust, erhöhte Latenz und Inkonsistenz ein. Die hohen Anforderungen an die Entwickler verlangen viel Vertrauen in jene, einer der zentralen Punkte des agilen Manifests [Michl 2018]. Insgesamt weist die Event-Driven Architecture also eine sehr hohe Agilität auf und ist damit besonders für moderne Software und ihre stetig wechselnden Anforderungen geeignet.

5 FALLSTUDIEN UND PRAXISBEISPIELE

Blah …

6 DISKUSSION

7 ZUSAMMENFASSUNG UND AUSBLICK

LITERATUR

Phil Bianco, Rick Kotermanski, and Paulo Merson. 2007. *Evaluating a service-oriented architecture.* Citeseer.

Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pål Krogdahl, Min Luo, and Tony Newling. 2004. *Patterns: service-oriented architecture and web services.* IBM Corporation, International Technical Support Organization New York, NY ….

David Garlan and Mary Shaw. 1994. *An Introduction to Software Architecture.* Technical Report CMU/SEI-94-TR-021. https://insights.sei.cmu.edu/library/an-introduction-to-software-architecture/ Accessed: 2025-Jan-2.

Ramakrishna Manchana. 2021. Event-Driven Architecture: Building Responsive and Scalable Systems for Modern Industries. *International Journal of Science and Research (IJSR)* 10 (01 2021), 1706–1716. https://doi.org/10.21275/SR24820051042

Thomas Michl. 2018. *Das agile Manifest – eine Einführung.* Springer Berlin Heidelberg, Berlin, Heidelberg, 3–13. https://doi.org/10.1007/978-3-662-57699-1_1

Hassaan Siddiqui, Ferhat Khendek, and Maria Toeroe. 2023. Microservices based architectures for IoT systems - State-of-the-art review. *Internet of Things* 23 (2023), 100854. https://doi.org/10.1016/j.iot.2023.100854

A CODE-BEISPIELE

A.1 Event-Driven-Architecture

```java
package demo.eda.service;

import demo.eda.event.OrderCreatedEvent;
import demo.eda.event.PaymentProcessedEvent;
import demo.eda.model.Payment;
import demo.eda.repository.PaymentRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.core.reactive.ReactiveKafkaProducerTemplate;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Flux;
import reactor.kafka.receiver.KafkaReceiver;
import reactor.kafka.receiver.ReceiverOptions;

import java.util.UUID;

@Service
@RequiredArgsConstructor
public class PaymentService {

    private final PaymentRepository paymentRepository;
    private final ReactiveKafkaProducerTemplate<String, PaymentProcessedEvent> producer;

    @KafkaListener(topics = OrderCreatedEvent.TOPIC, groupId = "payment-service")
    public Flux<Void> processPayment(ReceiverOptions<String, OrderCreatedEvent> receiverOptions) {
        return KafkaReceiver.create(receiverOptions)
                .receive()
                .flatMap(record -> {
                    final OrderCreatedEvent event = record.value();
                    final Payment payment = new Payment(UUID.randomUUID().toString(), event.getOrderId(), true);
                    return paymentRepository.save(payment).flatMap(savedPayment -> {
                        final PaymentProcessedEvent paymentEvent =
                            new PaymentProcessedEvent(
                                    savedPayment.getOrderId(),
                                    savedPayment.getId(),
                                    savedPayment.isSuccess());
                        return producer.send(PaymentProcessedEvent.TOPIC, paymentEvent).then();
                    });
                });
    }
}
```

Listing 1. Service-Implementierung des `PaymentService` in Java Spring Boot 3.4.1 mit Apache Kafka als Event-Broker

B ÜBUNGSAUFGABEN

B.1 Übungsaufgabe 1

Blah …